CLEMSON
School of COMPUTING

Union-find data structures
Tree based

Two efficient implementations

- List-based implementation (last time)
- Tree-based implementation (our focus today) ∈ Efficient

- A union-find data structure always involves three operations
  - MakeSet(e) – create a singleton with one element e
  - Union(A,B) – generate AUB from A and B
  - Find(e) – return the name of the set containing e

Tree-based. How does it work?

No longer a list

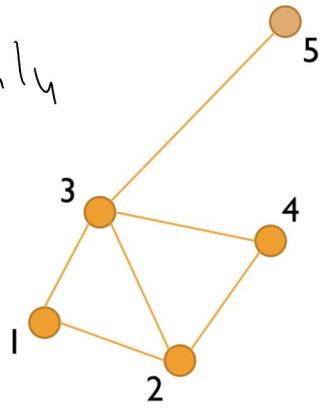
Class Node{

Value el //the element in the problem set
Node* ptr //pointer to the parent node

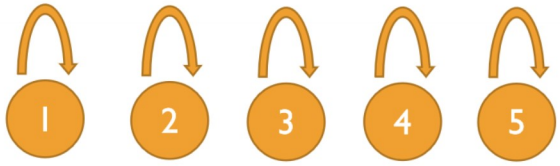
}

Tree only

No headers



MakeSet



← Default

value can be any data structure you want to group

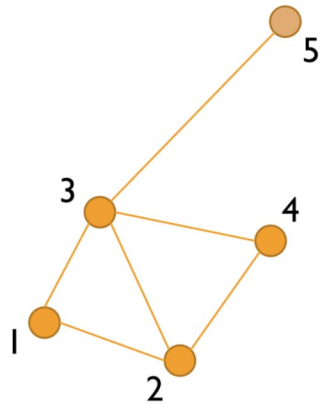
when pointing at self = group parent

Tree-based. How does it work?

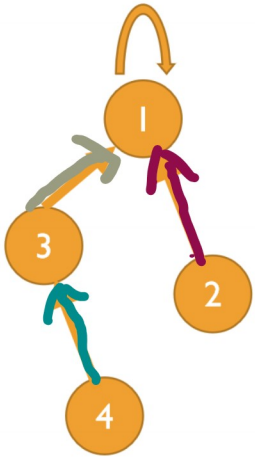
```
Class Node{

        Value el  //the element in the problem set
        Node* ptr //pointer to the parent node

}
```



Union(A,B)



- Union(1,2)
- Union(3,4)
- Union(1,3)

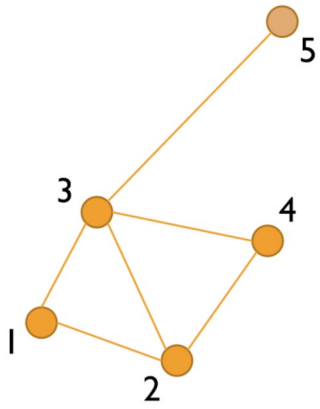
Union(Find(Node 1), Find(Node 2))

Tree-based. How does it work?

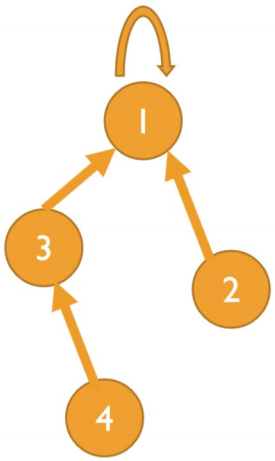
```
Class Node{

        Value el  //the element in the problem set
        Node* ptr //pointer to the parent node

}
```



Find(Node)



Find(4)

Tree-based. How to implement?

lesc efficiently

```
Class Node{
        Value el  //the element in the problem set
        Node* ptr //pointer to the parent node
}
```

V = value

node `Function makeSet(v){` O(c)

```
        Node node
        node.el = v
        node.ptr = node
}
```

pointing at self

void `Function union(A,B){` O(c)

```
        B.ptr = A
}
```

B root now pointing at A root

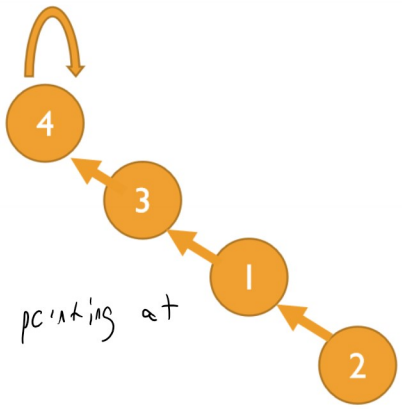
Node `Function find(node1){` O(n)

```
        Node node = node1
        while(node.ptr = node){
                node = node.ptr
        }
        return node
}
```

until at root →

, node = node pointing at

Return root

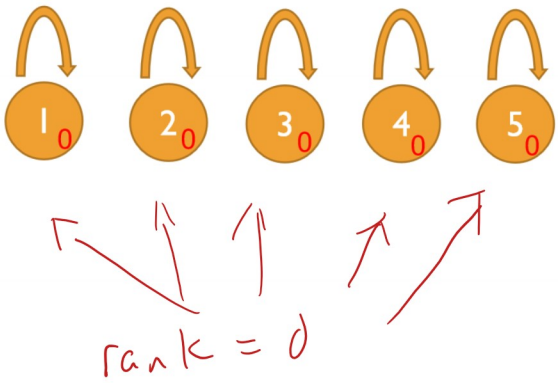


Tree-based. How to implement?

more efficiently

```
Class Node{
        Value el  //the element in the problem set
        Node* ptr //pointer to the parent node
        rank      //informally store the size of a tree
}
Function makeSet(v){
        Node node
        node.el = v
        node.ptr = node
        node.rank = 0
}
```

O(c)



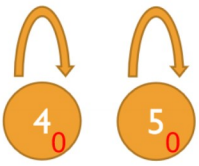
Tree-based. How to implement?

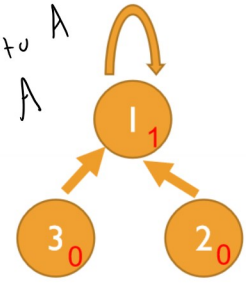
```
Class Node{
        Value el  //the element in the problem set
        Node* ptr //pointer to the parent node
        rank      //informally store the size of a tree
}

Function union(A,B){
        if(A.rank < B.rank){
                A.ptr = B
        }
        else{
                if(A.rank == B.rank){
                        A.rank = A.rank+1
                }
                B.ptr = A
        }
}
```

If A is likelier to be smaller, add A to B

If A = B, adding B to A levels up A





Tree-based. How to implement?

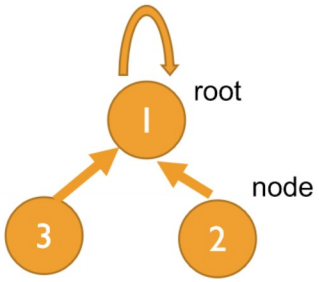
```
Function makeSet(v){                                    O(c)
        Node node
        node.el = v
        node.ptr = node
}
Function union(A,B){                                    O(c)
        if(A.rank < B.rank){
                A.ptr = B
        }
        else{
                if(A.rank == B.rank){
                        A.rank = A.rank+1
                }
                B.ptr = A
        }
}
Function find(node1){                                   O(logn)
        Node node = node1
        while(node.ptr = node){
                node = node.ptr
        }
        return node
}
```

Tree-based. Path-compression

```
Function find(node1){
        Node root = node1
        while(root.ptr = root){
                root = root.ptr
        }

        Node node = node1
        Node next
        while(node.ptr = root){
                next = node.ptr
                node.ptr = root
                node = next
        }

        return root
}
```



Path compression

- It was once believed that the amortized cost obtained was O(1)
- This has been proved not to be true but still it is better than O(logn)
- We need to introduce two new functions
  - A(m,n) is the Ackerman's function. This is the fastest growing function known.

$$
\begin{aligned}
A(1,j) &= 2^j \qquad \text{for } j \geq 1, \\
A(i,1) &= A(i-1,2) \qquad \text{for } i \geq 2, \\
A(i,j) &= A(i-1, A(i,j-1)) \qquad \text{for } i,j \geq 2.
\end{aligned}
$$

- Since the Ackerman function grows so fast, its inverse grows very slowly

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \lg n\}.$$

The important part for us is that, after initialization, any sequence of m union's and find's (using path compression) on an initial set of n elements can be performed in time O(mα(m,n)) time. Thus the amortized cost of each union-find operation is O(α(m, n)).

Tree-based implementation

- The tree-based implementation keeps track of the two sets with trees
  - Merging two trees is a very fast operation O(c)
  - Finding the root of the set a node belongs to is more involved.
    - With no optimizations this operation takes O(n)
    - With union-by-size we can decrease its complexity to O(logn)
    - With path-compression we can decrease it further to the inverse of the Ackerman function.

What's next?

- Book chapters
  - Chapter 7.3
- In class activities
  - Use tree-based union-find on new examples
- Next class
  - Tree data structures

Announcements

- Programming Assignment 1
  - Due on Feb 7th (next Tuesday)
  - Join the slack channel **#assignments** if you didn't
  - Go to TA's office hours:
    - Guoxi Liu
      - Monday 11 AM – 12 PM, McAdams Hall 110D
      - Tuesday 2 PM – 3 PM, McAdams Hall 109
    - Xueyi Bao
      - Wednesday 11 AM – 12 PM, McAdams Hall 110D
      - Thursday 1 PM – 2 PM, McAdams Hall 110B

Announcements

- Code examples used in this class:
  - https://github.com/clemson-cpsc3210/LectureExamples
  - Written in C++ language, built with CMake tool
  - Tested on Unix-like environments (i.e., Linux and macOS)
  - Raise issues on GitHub if you find any problems

Tree-based Union-Find Structure

Exercise

Suppose a sequence of m operations include MAKESET, UNION, or FIND where n of them are MAKESET ($m \geq n$).

(1) Both union-by-rank and path-compression are used, but UNION must appear before any FIND operation. Show that the worst-case amortized cost of the operation in this case is $O(1)$ instead of $O(\alpha(n))$.

Solution

(1) Both union-by-rank and path-compression are used, but UNION must appear before any of FIND operation. Show that the worst-case amortized cost of the set of operations in this case is $O(1)$ instead of $O(\alpha(n))$.

With n MAKESETs, there can be at most n – 1 UNION operations (since every UNION would reduce one set each time and there is at least 1 remaining set in the sequence of operations).

There can be at most n – 1 tree edges after n – 1 UNION. Note that FIND is performed after any of the Union operations, no path-compression is used before any UNION. At any time one FIND is executed, it will create edge from the node to the root while traversing upward from the node to the root, any following FIND performed on the node would take $O(1)$.

Since there are at most $n - 1$ tree edges in the disjoin-set tree (after $n - 1$ UNION), the total cost of $m - (2n - 1)$ FINDs will be $O\big(m - (2n - 1)\big) + \#\ of\ \ tree\ edges) \ = \ O(m - n) \ = \ O(m)$ since $n \ < \ m$.

Then, the total cost of the entire sequence obtained by add the n MAKESET and n – 1 UNION operations is $O(m) \ + \ O(n) \ = \ O(m)$. Therefore, the amortized cost will be $O(m)/m \ = \ O(1)$.

Exercise

Suppose a sequence of m operations include MAKESET, UNION, or FIND where n of them are MAKESET ($m \geq n$).

(1) Both union-by-rank and path-compression are used, but UNION must appear before any FIND operation. Show that the worst-case amortized cost of the set of operations in this case is $O(1)$ instead of $O(\alpha(n))$.

(2) Only path-compression is used, and UNION still must appear before any FIND operation. In this case, do we have a different worst-case amortized cost?

Solution

(2) Only path-compression is used, and UNION still must appear before any FIND operation. In this case, do we have a different worst-case amortized cost?

The answer is no. With tree-based data structure, the UNION still cost $O(1)$ and the largest possible tree edges is still $n - 1$ after $n - 1$ UNION operations. Thus, the largest possible cost for the first FIND would still be $O(\#\ of\ tree\ edges)$. After that, all the "related edges" will be redirected to the root and each following FIND operation will take $O(1)$.

The maximum possible number of UNION remains the same $n - 1$, the number of FIND is still $m - (2n - 1)$, the total cost does not change, amortized cost does not change.